Теория формальных языков и трансляций
Лекция 13: Монадические парсер-комбинаторы

В. С. Полозов

11 декабря 2013 г.

1 Парсеры с точки зрения ФП

Классическая статья: Monadic parsing in Haskell, `http://www.cs.nott.ac.uk/~gmh/pearl.pdf` [1].

Мы дадим немного отличающийся вариант, но не принципиально.

Реализация будет дана на языке Haskell, и будет пользоваться его ленивостью. А чистота языка потребует выведение всех зависимостей в явном виде.

1.1 Понятие парсера

Попробуем определить тип "парсер"как функцию, принимающую строку и возвращающую результат:

```
type Parser a = String -> a
```

Например, парсер для целых чисел можно было бы определить через библиотечную функцию read:

```
parseInt :: Parser Int
parseInt = read
```

У этого определения есть некоторые недостатки: оно показывает, что вернуть при успешном синтаксическом анализе, но что делать если рассматриваемая строка не принадлежит описанному языку? Можно добавить обработку исключительных ситуации, но это не лучший путь. Добавим тип, показывающий успешность разбора:

```
data ParseResult a = Ok a
                   | Fail String
  deriving Show
```

Где тип α - возвращаемый тип, строка в параметре конструктора Fail показывает причину, по которой разбор не успешен. Класс Show добавлен для удобства отладки.

Второе замечание к вышеописанному типу Parser – по этому определению строка разбирается целиком. Чтобы избежать этого будем в дополнение к результату возвращать непросмотренную часть входной строки:

```
newtype Parser a = P (String -> (String, ParseResult a))
```

Где, newtype и конструктор P добавлены для удобства и лучшей типизации. Лучше было бы конструктор назвать 'Parser', но выберем имя покороче, чтобы меньше писать.

Добавим функцию для применения парсера к строке:

```
apply :: Parser a -> String -> (String, ParseResult a)
apply (P p) s = p s
```

1.2 Примитивные парсеры

Рассмотри несколько примеров простых парсеров, реализующих введенный тип.

Самым простым примером, вероятно, будет парсер, который успешно принимает любой вход ничего не делая:

```
success :: Parser ()
success = P (\ s -> (s, Ok ()))
```

Рассмотрим пример парсера, который проверяет первый символ в строке на равенство заданному:

```
sym :: Char -> Parser Char
sym ch = P (\s -> f s)
    where
      f [] = ("", Fail "empty string")
      f s@(x:xs) | x == ch    = (xs, Ok x)
                 | otherwise = (s, Fail ("symbol not matched: '"++[ch]++"'"))
```

Примеры его использования в ghci:

```
*Main> apply (sym 'a') "abc"
("bc",Ok 'a')
*Main> apply (sym 'a') "bc"
("bc",Fail "symbol not matched: 'a'")
```

Парсер пустой строки может быть построен как сравнение строки, которая в языке Haskell является списком, с конструктором пустого списка.

```
eof :: Parser ()
eof = P f
  where
    f [] = ("", Ok ())
    f s = (s, Fail "not eof")
```

1.3 Комбинаторы парсеров

Следующая задача: научиться комбинировать примитивные парсеры. Например, по паре парсеров построить их последовательность. Другим примером может быть альтернатива.

Определим последовательность как инфиксную левоассоциативную операцию |>:

```
infixl 7 |>
```

Последовательность принимает два парсера, и действует следующим образом: сначала применяем первый к входной строке, если его результат успешен, то применяем второй к оставшейся непросмотренной части входной строки.

```
p1 |> p2 = P f
  where
    f s = case apply p1 s of
          (_, Fail r) -> (s, Fail r)
          (s', Ok _) -> apply p2 s'
```

Например, парсер «sym 'a' |> sym 'b'» разбирает строку "ab":

```
*Main> apply (sym 'a' |> sym 'b') "abc"
("c",Ok 'b')
*Main> apply (sym 'a' |> sym 'b') "cde"
("cde",Fail "symbol not matched: 'a'")
*Main> apply (sym 'a' |> sym 'b') "ac"
("c",Fail "symbol not matched: 'b'")
```

А парсер для строки "abc"целиком, можно записать как «sym 'a' |> sym 'b'|> sym 'c' |> eof»:

```
*Main> apply (sym 'a' |> sym 'b' |> sym 'c' |> eof) "abc"
("",Ok ())
*Main> apply (sym 'a' |> sym 'b' |> sym 'c' |> eof) "abcd"
("d",Fail "not eof")
```

По аналогии сделаем альтернативу: бинарная операция с меньшим приоритетом, принимающая пару парсеров. Если первый успешен, то принимаем, если нет – то запускаем второй парсер.

```
infixl 6 <|>

(<|>) :: Parser a -> Parser a -> Parser a
p1 <|> p2 = P f
  where
```

```
    f s = case apply p1 s of
            (s', Ok r) -> (s', Ok r)
            (_, Fail r) -> apply p2 s
```

Что работает ожидаемым образом:

```
*Main> apply (sym 'a' <|> sym 'b') "ac"
("c",Ok 'a')
*Main> apply (sym 'a' <|> sym 'b') "bc"
("c",Ok 'b')
```

И даже в паре с последовательностью:

```
*Main> apply (((sym 'a' |> sym 'b') <|> (sym 'a' |> sym 'c')) |> sym 'd') "acd"
("",Ok 'd')
*Main> apply (((sym 'a' |> sym 'b') <|> (sym 'a' |> sym 'c')) |> sym 'd') "abd"
("",Ok 'd')
```

Однако, такое определение альтернативы, ожидаемо работает жадно: если первая альтернатива успешна, то вторая не будет применена независимо от того, что случится далее. Например, если обе альтернативы подходят, но первая из них далее приводит к ошибке:

```
*Main> apply (((sym 'a' |> sym 'b') <|> sym 'a') |> sym 'b' |> sym 'c') "abc"
("abc",Fail "symbol not matched: 'b'")
*Main> apply ((sym 'a' <|> (sym 'a' |> sym 'b')) |> sym 'b' |> sym 'c') "abbc"
("bc",Fail "symbol not matched: 'c'")
```

1.4 Возвраты при разборе

Как вариант решения проблемы жадной альтернативы можно предложить следующий: изменим тип Parser так, чтобы возвращался не один результат, а все возможные:

```
newtype Parser a = P ( String -> [(String, ParseResult a)] )
```

В таком случае альтернатива определяется как конкатенация списков:

```
(<|>) :: Parser a -> Parser a -> Parser a
p1 <|> p2 = P f
  where
    f s = apply p1 s ++ apply p2 s
```

Соответствующим образом надо изменить определения примитивных парсеров, так, чтобы они возвращали списки. Например:

```
sym ch = P f
  where
    f [] = [("", Fail "empty string")]
    f s@(c:s') = if c == ch
                 then [(s', Ok ch)]
                 else [(s, Fail "not matched")]
```

Заметим, что пустой список эквивалентен ошибке – в получившихся результатах не было ни одного удовлетворяющего. Соответственно, напишем функцию оставляющую из результатов только успешные:

```
choose :: [(String, ParseResult a)] -> [(String, ParseResult a)]
choose rs = filter (\(_, r) -> case r of
                                 Ok _ -> True
                                 Fail _ -> False)
                   rs
```

Тогда последовательность выражается следующим образом:

```
(|>) :: Parser a -> Parser b -> Parser b
p1 |> p2 = P f
  where
    f s = case choose $ apply p1 s of
            [] -> [(s, Fail "")]
            rs -> concat $ map (\(s', _) -> apply p2 s') rs
```

Замечание. Приведенная реализация choose теряет информацию об ошибках, если таковая была. Можно было бы реализовать её следующим образом: если разбор успешен, то вернуть все варианты успешного разбора, иначе – вернуть самую «далекую» ошибку, т.е. с максимально длинным префиксом.

Список литературы

[1] Graham Hutton and Erik Meijer. Monadic parsing in haskell. J. Funct. Program., 8(4):437–444, July 1998.